Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU

Kinshuk Govil* Edwin Chan† Hal Wasserman‡

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

To take advantage of the full potential of ubiquitous computing, we will need systems which minimize power-consumption. Weiser *et al.* and others have suggested that this may be accomplished by a CPU which dynamically changes speed and voltage, thereby saving energy by spreading run cycles into idle time. Here we continue this research, using a simulation to compare a number of policies for dynamic speed-setting. Our work clarifies a fundamental power vs. delay tradeoff, as well as the role of prediction and of smoothing in dynamic speed-setting policies. We conclude that success seemingly depends more on simple smoothing algorithms than on sophisticated prediction techniques, but defer to the replication of these results on future variable-speed systems.

1 Introduction

Recent developments in ubiquitous computing make it likely that the future will see a proliferation of cordless computing devices. Clearly it will be advantageous for such devices to minimize power-consumption. The top power-consumers in a computer system are the display (68%), the disk (20%), and the CPU (12%) [6]. There is seemingly little which can be done to minimize screen power-consumption, beyond employing a screen-saver and waiting for hardware improvements. Disk power-consumption may be minimized by spinning down the disk when it has been inactive for several seconds; [3, 6, 7] have researched this topic.

*kinshuk@csua.berkeley.edu.
†chance@cory.eecs.berkeley.edu.
‡halw@cs.berkeley.edu. Supported by NDSEG Fellowship DAAH04-93-G-0267.

MOBICOM 95 Berkeley CA USA
© 1995 ACM 0-89791-814-2/95/10..$3.50

In the future we may well see ubiquitous computing devices with neither disks nor conventional displays; and, for such devices, minimizing the power-consumption of the CPU will be particularly critical. Methods for saving CPU power have been suggested by [2, 4, 9]. They point out that some CPUs can run at a range of possible speeds: and voltage may then be decreased as speed decreases. From [9] we inherit the assumption that, over an idealized device's voltage range $[V_{min}, V_{max}]$, voltage may be decreased in direct proportion to speed: this is a valid first-order approximation [1]. Now, a CPU, regarded as a capacitor-based system, satisfies the physical law

$$\text{energy/sec} \propto \text{voltage}^2 \cdot \text{speed}$$

or equivalently

$$\text{energy/task} \propto \text{voltage}^2.$$

And so it is possible to save on overall energy-usage by reducing voltage. In particular, if voltage may be reduced in direct proportion to speed, then

$$\text{energy/task} \propto \text{speed}^2.$$

Hence it would be advantageous to have a CPU capable of dynamic speed-setting. Such a CPU could well decrease power-usage without inconvenience to the user. For example, a CPU might normally respond to a user's command by running at full speed for 0.001 seconds, then waiting idle; running at one-tenth speed, the CPU could complete the same task in 0.01 seconds, thereby saving energy without generating noticeable delay.

Essential performance factors of a dynamic speed-setting policy are **power-savings** and **delay**. To save power, a CPU would ideally run at a flat, average speed. But this would result in unacceptable delay; hence a tradeoff between the two factors must be accomplished. The question of how to measure delay is found to be non-trivial, as is the question of how much delay is acceptable. Ideally, we would know the maximum allowable delay for each process; but in existing systems such information is not generally available.

In seeking to strike an optimal balance between low power-consumption and low delay, an algorithm must consider issues of **prediction** and **smoothing**. Due to pragmatic limits on the frequency with which CPU speed can be changed, a speed-setting policy must first predict how busy the CPU will be in the near future. Then, given this prediction, the policy must make a decision aimed at smoothing speed. For example, if a peak in CPU usage is predicted, the policy might increase speed, but it might also keep speed low, thereby evening out speed at the cost of increasing delay.

The distinction between prediction and smoothing is somewhat subjective. For instance, a speed-setting algorithm which strongly attempts to set a flat, average speed may be thought of in terms of prediction (it always predicts that the future will be like the average) or in terms of smoothing (it smoothes to the greatest extent possible). Nevertheless, we will attempt to separate the two functions to some extent, trying to understand the utility (or lack thereof) of several algorithms for prediction and for smoothing.

Weiser *et al.* [9] present only one practical speed-setting policy, PAST. PAST's prediction algorithm is elementary, and its smoothing somewhat ad hoc. Hypothesizing that more sophisticated prediction methods will allow for substantially improved performance, we here set out to test several new policies.

In Section 2, we review the simulation model employed by Weiser *et al.* In Section 3, we indicate how we have altered this model. In Section 4, we present a number of speed-setting policies. In Section 5, we analyze the performance of these policies. Finally, in Section 6, we present our conclusions and suggest avenues for future research.

2 Previous work

From Weiser *et al.* [9] we inherit a simulator, and with it a number of assumptions.

Only the CPU's energy-usage is studied; there is no consideration of the energy costs which CPU-slowing could generate elsewhere in the system. Indeed, due to the radically different time-frames involved, it is reasonable to focus on the CPU alone. For example, CPU-slowing could stretch a 0.001 second task to 0.01 seconds, whereas disk spindown might typically take place after 2–300 seconds of disk idle [6]; hence we may well expect that the former will have no substantial effect on the latter. In order to support this argument, our policies will explicitly limit delay to a brief interval-length.

The simulator takes as input traces of CPU-usage for a workstation running standard applications. No attempt is made to capture the unique workload (if any) of a ubiquitous computing device.

Trace data is first divided into uniform-length **time intervals**. For each interval, one computes the **run_percent**: the fraction (on range [0, 1]) of cycles in which the CPU is active. Figures 1 and 2 give examples of such data for **interval_lengths** 0.01 seconds and 0.05 seconds. Not surprisingly, the run_percent values are more bursty for the smaller interval_length.

It is assumed that, for a given interval, speed may be set to any real number on range [min_speed, 1], where 1 represents full speed. Weiser *et al.* compile data for min_speeds 0.2, 0.44, and 0.66 (corresponding to idealized CPUs with full voltage 5.0 V and minimum voltages 1.0 V, 2.2 V, and 3.3 V). It is also assumed that speed changes have no time or energy cost.

Moreover, it is assumed that no energy is expended during intervals in which no work is done; i.e., the ability to dynamically switch the CPU on/off is implicit. Such a capability is indeed technically feasible: for example, the Intel Pentium Processor can, in ≤ 50 μsec, enter a low-current sleep state in which power-usage drops 85% [5, Section 5.3]. In the energy-usage graphs of Weiser *et al.* and in our own, the maximum energy level (energy usage = 1) already reflects savings due to a zero-power sleep state. The extent of this initial saving is not here studied, but may well be substantial.

Each Weiser *et al.* simulation runs a policy on trace data ten times, using interval_lengths 0.001, 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds. When, during each run, speed is not fast enough to complete an interval's work, **excess_cycles** spill over into later intervals. To summarize the simulation results, energy-usage is plotted as a function of interval_length. The simulator also computes **delay_penalty**, a somewhat subjective measure of total delay determined by reference to all values of excess_cycles.

If an optional flag is set, the simulator attempts to divide idle time into **soft_idle**, into which run_cycles may allowably be stretched, and **hard_idle**, which must be left intact. For example, it is valid to stretch run_cycles into time spent waiting for the user's next command (soft_idle), but not valid to stretch a process into time during which it must wait after requesting data from disk (hard_idle).

The Weiser *et al.* policies may change CPU speed only at the start of those intervals containing a process-start or process-stop event. Thus speed may not be recomputed for intervals in the midst of long runs or long idles.

The simulator cannot model event reordering due to speed changes, and cannot identify situations in which delays could be invidiously additive. This seems an inherent difficulty of simulating on limited trace data.

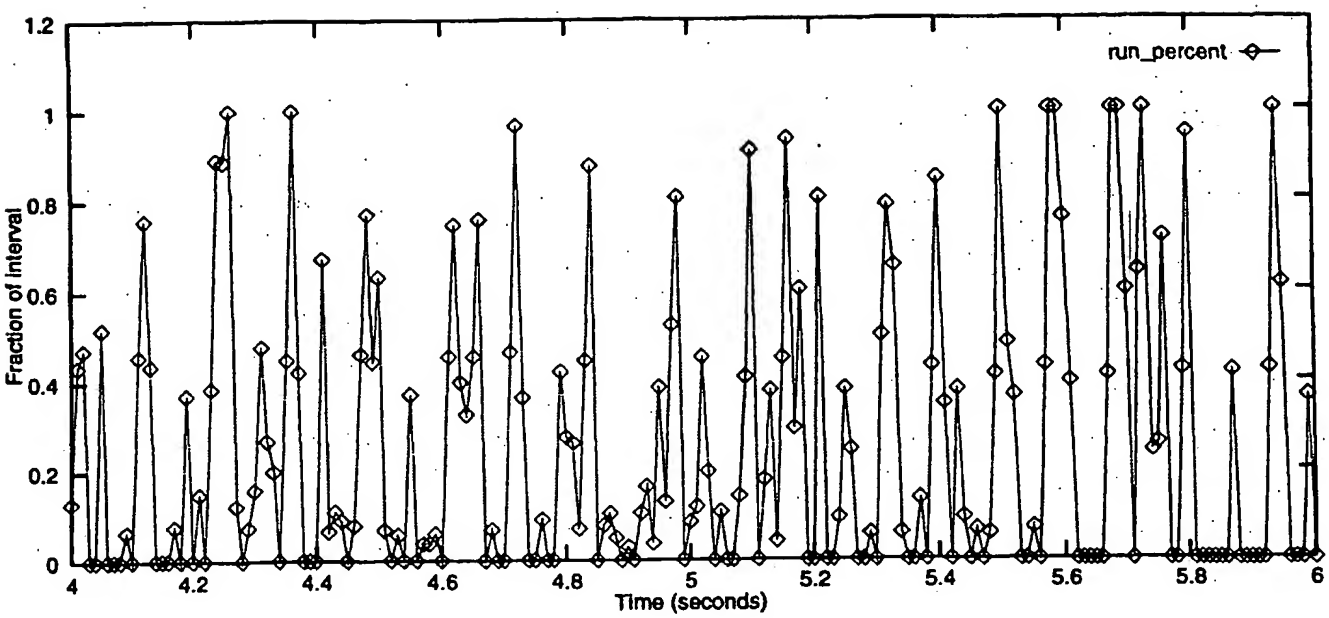

Figure 1: Run_percent values. Trace emacs1, interval_length 0.01 seconds.

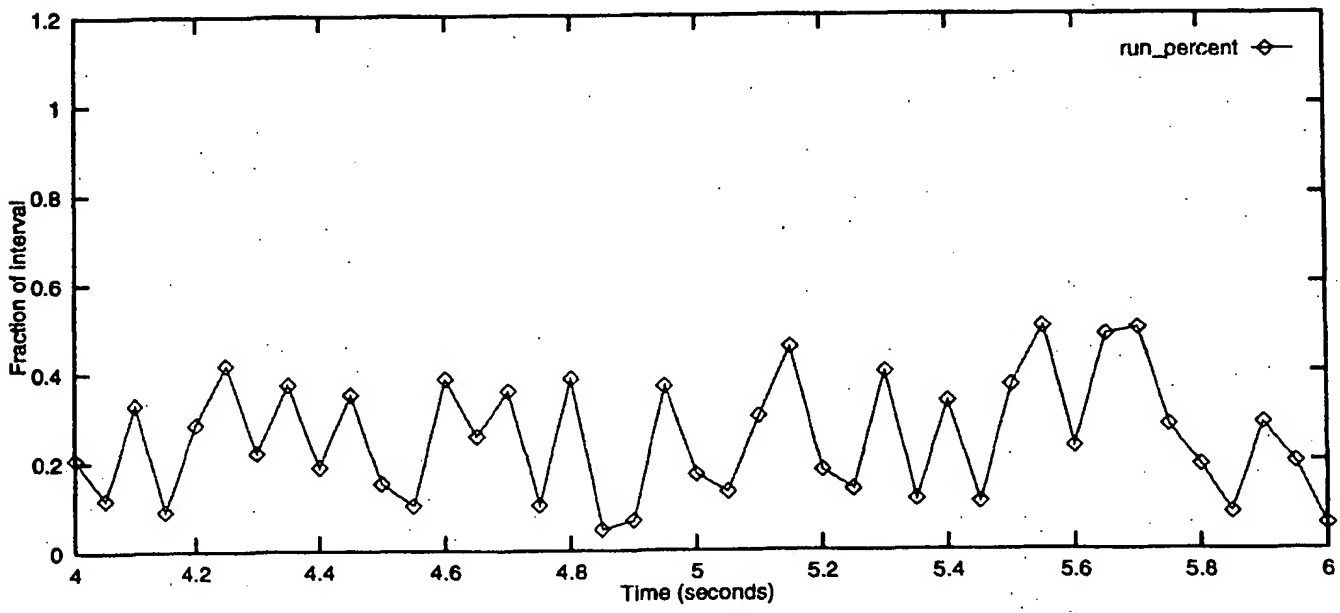

Figure 2: Run_percent values. Trace emacs1, interval_length 0.05 seconds.

3 Simulation model

We employ the same traces used by Weiser *et al.* Our simulator is based on theirs, but has been changed in several respects.

- All of our simulations are for min_speed = 0.2.
- To speed up the simulation, we did not employ the 0.001 second interval_length.
- An error was corrected: due to a programming bug, the Weiser *et al.* simulator was overly optimistic about the amount of work which could be completed in certain intervals.[1]
- The Weiser *et al.* option of dividing idle time into hard_idle and soft_idle seemed degenerate, often failing to identify any significant amount of soft_idle. Most of our simulations are therefore run without the hard/soft option. However, we later confirm that runs with the hard/soft option yield similar results.
- We recompute speed at the beginning of each interval, even if the interval is in the midst of a long run or idle. This policy could be regarded as less efficient than that employed by Weiser *et al.*, as its implementation would require additional interrupts. On the other hand, we feel the Weiser *et al.* model to be unrealistic: for it gives its speed-setting policy premature knowledge that a long run has begun. Moreover, we wished to create and analyze policies in a way more suited to a uniform, per-interval speed-setting.[2]
- Rather than plotting our results as power vs. interval_length, we plot power vs. a delay measure. Thus we attempt to focus more clearly on the power vs. delay tradeoff, regarding interval_length as a merely internal parameter.

In theory the two plotting methods are related, as the Weiser *et al.* PAST policy attempts to limit delay to an interval_length. However, as excess_cycles are allowed to spill over into future cycles, true delay is actually an unclear function of interval_length and delay_penalty. The importance of delay_penalty is clearly indicated by the fact that Weiser *et al.*'s FUTURE, an artificial policy which has perfect knowledge of the coming interval, nevertheless uses *more energy* than PAST because it is not allowed to transfer excess_cycles between intervals.

Instead of trying to combine interval_length and delay_penalty into a meaningful composite number, we substituted our own measure of delay, which is illustrated in Figure 3. For a given CPU task, imagine plotting the amount of work that remains to be done as a function of time. The lower line in Figure 3 corresponds to the CPU running at full speed; the upper line illustrates the same task being run slowly and intermittently on a variable-speed CPU. We take the area between the two lines as a measure of the task's delay; we then divide the sum of all the "delay areas" in the trace by the sum of all the "full speed areas" to derive a figure for average delay. This measure, while still arbitrary, has some sophistication: it fairly represents delay both within and between intervals, it is not unduly sensitive to arbitrary time-slicing of tasks into smaller tasks, and it considers a task which has nearly been completed before a delay (and so may already have generated child processes or critical portions of output) as having a lower delay figure that a task delayed at its start.

4 Speed-setting policies

4.1 PAST

PAST is the only practical speed-setting policy proposed by Weiser *et al.* We employ it for purposes of comparison.[3]

- PAST:
  - **Prediction:** PAST calculates how busy the last completed interval was (including excess_cycles brought into that interval). It then predicts that the coming interval will be equally busy.
  - **Speed-setting:** If the prediction is for a busy interval, PAST increases speed; if for a mostly idle interval, PAST decreases speed. Some smoothing is accomplished by limiting the amount by which speed can change (except that speed may be increased to 1 if excess_cycles rises particularly high).[4]

Since PAST attempts to complete work within the interval after that which generated it, it is not surprising that delay rises and power-usage falls as interval_length increases. Weiser *et al.* identify the range of interval_lengths from 0.01 seconds to 0.05 seconds as one in which delay and energy-savings both seem acceptable. For a simple algorithm, PAST does surprisingly well.

[1] This bug may be observed in the "Speed Setting Algorithm" of [9, Section 6]. In the second assignment statement, excess_cycles from the previous interval are added into run_cycles. In the fourth assignment statement, the time available to do work in the current interval is then given as (run_cycles + soft_idle); and the adjusted rather than raw value of run_cycles is mistakenly used here. Hence results may be substantially over-optimistic when excess_cycles is high.

[2] We also use this revised simulation method when running Weiser *et al.*'s policy PAST for purposes of comparison. We intend this as a measure friendly to PAST, as its performance improves due to this modification (as measured by the revised delay metric described below).

[3] It should be noted, however, that PAST is evidently only intended as a reasonable first-version policy. Moreover, our correction of a simulator bug, as noted in Section 3, has affected the performance of the policy in ways to which Weiser *et al.* did not have the opportunity to respond.

[4] Refer to [9] for full details.

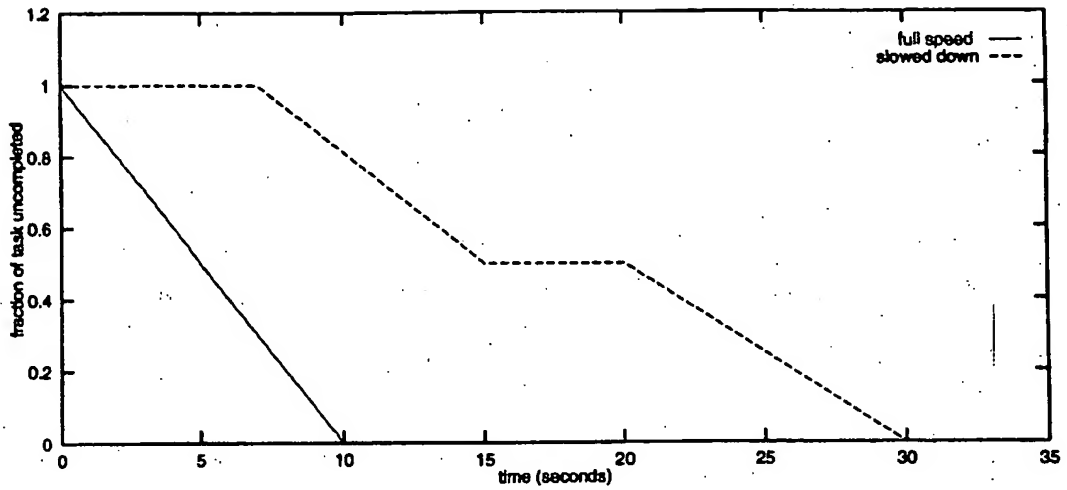

Figure 3: Graph of run-cycles not yet completed as a function of time, for a hypothetical process that normally takes 10 seconds to execute. The bottom line is for process execution at full speed; the top line, for process execution at a slower, intermittent speed.

Nevertheless, we felt that there was room for improvement here. We disagree with PAST's prediction algorithm: in a bursty trace such as that illustrated in Figure 1, the assumption that adjacent intervals will be similar is in fact almost certainly wrong. Moreover, PAST considers only the excess_cycles that went into the last completed interval, ignoring the seemingly more valuable figure of excess_cycles being pushed out from that interval into the coming interval. Since it looks back only one interval, PAST smoothes speed poorly; and the attempt to patch this problem with an arbitrary limit on speed-change seems ad hoc—and dangerous, in that a process can be delayed several intervals while the system slowly banks up speed. The behavior of PAST can be strange: given uniform input data, it can thrash speed without coming to a limit; and, due to mistaken speed-setting decisions, it saves little more energy when min_speed is 0.2 than when it is 0.44. Finally, we feel that the role of interval_length is confused, as it influences the outcome of the simulation in three different ways, determining *(1)* the frequency with which speed can be changed, *(2)* the acceptable amount of delay, and *(3)* how far back PAST looks when making its predictions. We would wish to see a clearer separation of these distinct functionalities.

In short, we felt that it should be possible to create stronger policies with more sophisticated prediction heuristics.

4.2 FLAT

Our first policy is FLAT. Weak on prediction, it simply tries to smooth speed to a global average. FLAT takes an input parameter ⟨*const*⟩, which must be a real number on range [0, 1].

- FLAT ⟨*const*⟩:
  - **Prediction:** Predict the new run_percent to be ⟨*const*⟩.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work *plus* the excess_cycles being pushed into the coming interval (subject, of course, to the limit of full speed = 1).

While FLAT wants to keep run_percents as flat as possible, it also responds effectively to excess_cycles. Indeed, speed is always set fast enough to complete at least the excess_cycles, so no work may be delayed more than one interval. The same speed-setting rule will be employed in most of our policies.

4.3 LONG_SHORT

LONG_SHORT is a more predictive policy, one which attempts to find a golden mean between local behavior and a more long-term average. Parameter ⟨*const*⟩ is a non-negative real number which, as it is increased, gives more weight to local behavior. We hoped to discover an optimal value of ⟨*const*⟩ at which LONG_SHORT would predict accurately by giving the best possible weight to local behavior. This may alternatively be thought of in terms of smoothing: LONG_SHORT attempts to smooth to a global average, but shows some respect for local peaks.

- LONG_SHORT (*const*):
  - **Prediction:** Look up the last 12 run_percents (and, for this policy, we include in each run_percent the excess_cycles pushed into the interval). The 3 most recent run_percents constitute the short-term past; the remaining 9, the long-term past. Our prediction for the coming run_percent is then a weighted sum of these 12 values, where each short-term value is given weight (*const*), each long-term value weight 1.
  - **Speed-setting:** Set speed fast enough to complete the predicted work.

For example, if (*const*) = 4 and the last 12 run_percents, including excess_cycles, are $0 \rightarrow .3 \rightarrow .5 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow .8 \rightarrow .5 \rightarrow .3 \rightarrow .1 \rightarrow 0 \rightarrow 0$, then we would set speed to

$$\frac{0+.3+.5+1+1+1+.8+.5+.3+4(.1+0+0)}{9+4(3)} = 0.276.$$

Note that LONG_SHORT, an early policy, is less elegant than FLAT and more like PAST, particularly in that the speed it sets is affected by excess_cycles only indirectly.

4.4 AGED_AVERAGES

A cleaner variant of LONG_SHORT, AGED_AVERAGES employs an exponential-smoothing method, attempting to predict via a weighted average: one which geometrically reduces the weight given to each previous interval as we go back in time. Parameter (*const*), a real number on range (0, 1], determines the rate of geometric reduction.

- AGED_AVERAGES (*const*):
  - **Prediction:** The predicted new run_percent is a weighted average of all previous run_percents, where the weight given to an interval's data is multiplied by a factor (*const*) for each 0.01 seconds that we go back in time.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

For example, if interval_length is 0.01 seconds, (*const*) is $\frac{2}{3}$, and the previous run_percents are P_t, P_{t-1}, $P_{t-2}, \ldots$, then the predicted new run_percent would be

$$\frac{1}{3} \cdot P_t + \frac{2}{9} \cdot P_{t-1} + \frac{4}{27} \cdot P_{t-2} + \cdots.$$

Note that (*const*) is defined in such a manner that aging will be essentially independent of interval_length; this we regard as a step toward reducing the confused multiple effects of interval_length.

We hoped that AGED_AVERAGES, like LONG_SHORT, would work best for a particular value of (*const*) at which it would optimally balance the long-term and the short-term past.

4.5 CYCLE

We now experiment with more sophisticated prediction algorithms. The CYCLE policy was inspired by run_percent plots such as Figure 2. Observe that these run_percent values look quite cyclical. Can we take advantage of such cycling to predict?

- CYCLE (*const*):
  - **Prediction:** Examine the last 16 run_percents. Does there exist $X \in \{1, 2, \ldots, 8\}$ such that the last $2X$ values approximately repeat a cycle of length X? If so, predict by extending this cycle. Or, if no good cycle is found, predict the new run_percent to be a flat (*const*).
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

To be more exact, we evaluate potential cycles by computing an error-measure equal to the mean average of the differences between all pairs of run_percents which should match. For example, say that the last eight run_percents are $0 \rightarrow .4 \rightarrow .8 \rightarrow .1 \rightarrow .3 \rightarrow .5 \rightarrow .7 \rightarrow 0$. Then an alleged cycle of length 4 would correspond to the following matchup:

0	.4	.8	.1
.3	.5	.7	0

and our error-measure for this matchup would be

$$\frac{|0-.3| + |.4-.5| + |.8-.7| + |.1-0|}{4} = 0.15.$$

We then predict according to the cycle with the lowest error-measure (if our example length-4 cycle was found to be best, our predicted new run_percent would be .3); or, if every potential cycle has error-measure > 0.2, we predict run_percent to be (*const*).

Observe that CYCLE behaves like FLAT except when it prefers to make a "smarter" guess by reference to a discovered cycle.

4.6 PATTERN

CYCLE is generalized in PATTERN, which employs a method reminiscent of branch prediction tables. Here we attempt to identify the most recent run_percent values as repeating a pattern seen earlier in the trace.

Let an interval of type A be one in which run_percent is between 0 and 0.25; type B, between 0.25 and 0.5; type C, between 0.5 and 0.75; type D, between 0.75 and 1. Then if, for example, the most recent run_percents are $0 \rightarrow .13 \rightarrow .28 \rightarrow .33 \rightarrow .52 \rightarrow .79$, this would correspond to pattern $AABBCD$. If, looking back in time, we find that the last occurrence of $AABBCD$ was followed by a fall back to A, we might guess that the same is about to happen now (and so would predict run_percent = 0.125—the middle of the A range).

Parameter ⟨*const*⟩ is a positive integer determining the pattern-length which PATTERN will employ.

- PATTERN ⟨*const*⟩:
  - **Prediction:** Convert the ⟨*const*⟩ most recent run_percents into a pattern in alphabet {A, B, C, D}. Then continue backward in time until you find find another occurrence of this pattern. Predict that the coming run_percent will have the same magnitude-letter as that which followed the previous instance of the pattern. Predict run_percent to fall in the middle of that letter's associated range ($A \Rightarrow 0.125$, $B \Rightarrow 0.375$, $C \Rightarrow 0.625$, $D \Rightarrow 0.875$).
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

This model of pattern-discovery is evidently partial and arbitrary. Nevertheless, we hoped that, when ⟨*const*⟩ was set optimally, repeated patterns—e.g., peaks of a certain common width—would be picked out to good effect.

4.7 PEAK

PEAK is a more specialized version of PATTERN. It uses heuristics based on the expectation of narrow peaks, such as those which occur frequently in Figure 1. Hence we expect rising run_percents to fall symmetrically back down, falling run_percents to continue falling, a sustained run_percent = 1 to fall but a sustained 0 to hold steady.

- PEAK ⟨*const*⟩:
  - **Prediction:** Let P_{t-1}, P_t be the two most recent run_percents, and let P_{t+1} be our prediction for the coming run_percent.
    - If $P_t > P_{t-1}$, then $P_{t+1} := \max\{P_{t-1}, 0.1\}$.
    - If $P_t < P_{t-1}$, then $P_{t+1} := \min\{P_t, 0.1\}$.
    - If $P_t = P_{t-1}$, then, if $P_t = 1$, $P_{t+1} := 0.4$; otherwise, $P_{t+1} := P_t$.
  - **Speed-setting:** Set speed fast enough to complete the expected new work plus a ⟨*const*⟩ fraction of excess_cycles.

Observe that we have modified our usual speed-setting policy of always trying to complete the expected new work plus all excess_cycles; this policy was logical but perhaps too cautious. Parameter ⟨*const*⟩ may now be varied from 1, indicating that we employ our usual speed-setting policy, to 0, indicating that we ignore the excess_cycles entirely.

5 Performance of our policies

In Section 5.1, we test each of our policies in turn, comparing them to PAST and finding the optimal value for each policy's ⟨*const*⟩ parameter. Runs are on trace emacs1, a relatively short trace of typing into an emacs buffer. In Section 5.2, we then evaluate the relative merits of our policies, and double-check with runs on a substantially different trace.

5.1 Runs of each policy

Figure 4 graphs the performance of FLAT running with several possible values of its ⟨*const*⟩ parameter; PAST is also provided for comparison. Observe that, for each policy, results are presented for a selection of nine interval_lengths; these generally form a curve, slanting toward more delay and less energy-usage as interval_length increases. The optimal algorithm that works within a given delay limit is found by starting on the x-axis at the desired delay figure and moving vertically until one reaches the lowest curve. Thus policies whose results curve closer to the origin are superior, while sets of data points which seem "shifted" along a single curve represent different ranges of possible energy-usage and delay but a similar energy vs. delay tradeoff.

We observe from Figure 4 that FLAT achieves optimality when ⟨*const*⟩ ≈ 0.4. We also note that FLAT is superior to PAST for all possible values of ⟨*const*⟩.[5] Indeed, we will find this to be essentially true for all of our policies.

Figure 5 graphs the performance of LONG_SHORT. In comparison to FLAT, this policy is shifted toward lower energy-usage and higher delay, possibly because its speed-setting is less responsive to excess_cycles. Results improve as ⟨*const*⟩ increases, leveling out around ⟨*const*⟩ = 3 to 5. This seemingly indicates that prediction based on the short-term past is particularly advantageous.

[5] This is not an artifact of our new delay measure. If one uses Weiser *et al.*'s delay_penalty instead, the difference is only more pronounced.

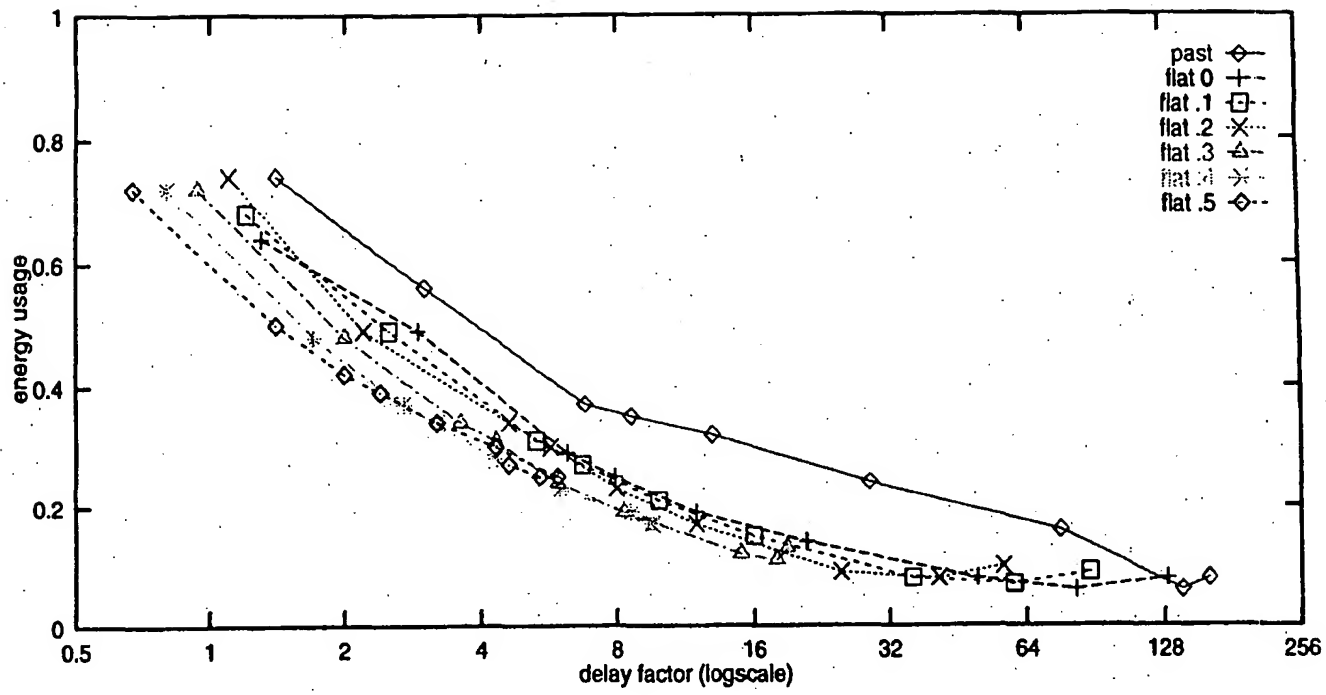

Figure 4: Performance of policy FLAT on trace emacs1. FLAT, run with several possible values of its parameter (*const*), is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

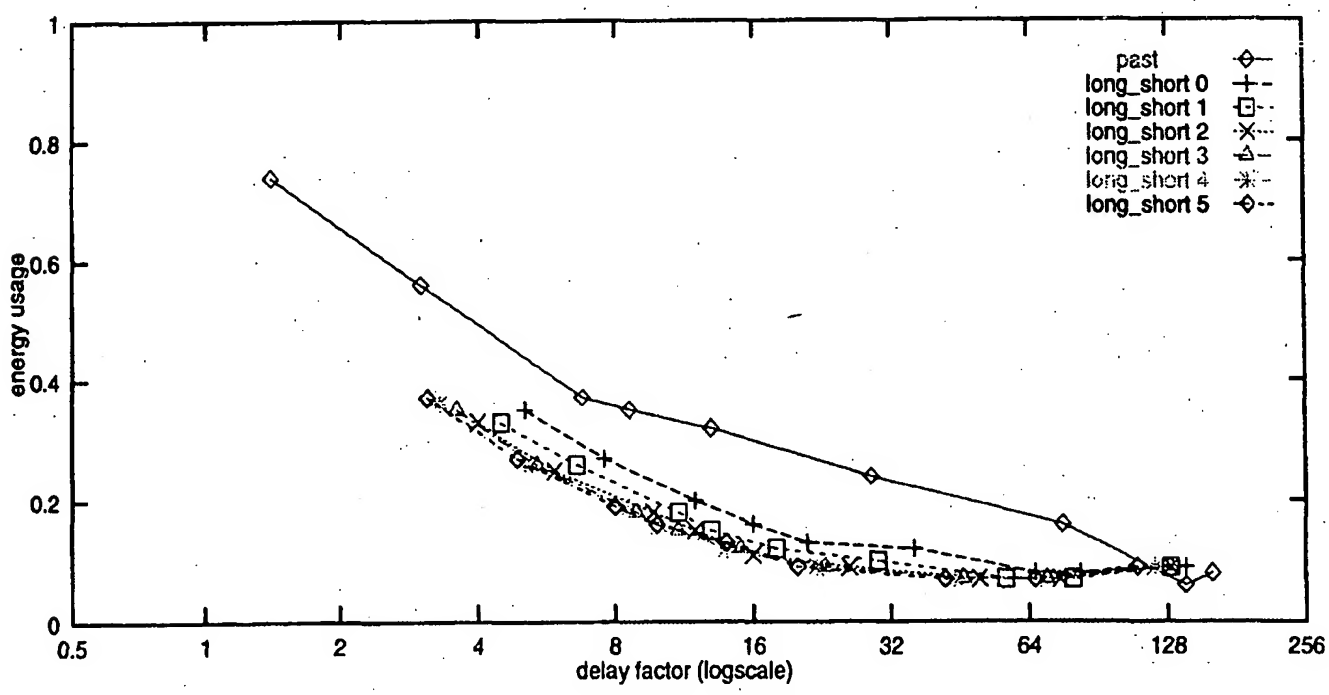

Figure 5: Performance of policy LONG_SHORT on trace emacs1. LONG_SHORT, run with several possible values of its parameter (*const*), is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

Figure 6 graphs the performance of AGED_AVERAGES. We consider this result a disappointment because, rather than indicating an optimal value for $\langle const \rangle$, performance merely improves as $\langle const \rangle$ increases. Thus AGED_AVERAGES works best when $\langle const \rangle = 1$, in which case it simply predicts by calculating the unweighted average of all run_percents so far. Thus, it seems that the best one can do here is to predict via a global average, in which case AGED_AVERAGES is little different from FLAT.

Figure 7 graphs the performance of CYCLE. Results seem lackluster; $\langle const \rangle \approx 0.5$ is optimal.

Figure 8 graphs the performance of PATTERN. The results here are particularly disappointing: there is no significant change as $\langle const \rangle$ varies, suggesting the meaningful patterns are not being found.

Figure 9 graphs the performance of PEAK. Results seem strong, with PEAK achieving optimality when $\langle const \rangle \approx 0.2$. Note, however, that PEAK performs little better when $\langle const \rangle = 0.2$ than when $\langle const \rangle = 1$: there is—not surprisingly—a shift toward greater delay and less energy-usage, but the energy vs. delay tradeoff is not improved. This indicates that our experiment of being lazier about the completion of excess_cycles has had little effect. Thus, if PEAK proves to be a strong policy, we should attribute this to the prediction algorithm rather than to our experiment in speed-smoothing.

5.2 Comparing the policies

Figure 10 summarizes the performance of the best policies from Figures 4–9. Surprisingly, the simplest policy, FLAT 0.4, is optimal for delay values ≤ 8, while LONG_SHORT 3, which is scarcely more complex, is optimal for the higher delay values. Of our more sophisticated predicting algorithms, PEAK 0.2 does best, coming close to equaling FLAT and LONG_SHORT in the medium-delay range. AGED_AVERAGES, CYCLE, and PATTERN are all disappointing. It is particularly telling that CYCLE is consistently worse than FLAT. For CYCLE imitates FLAT except when it is "trying to be clever"; and so this result would suggest that when CYCLE tries to be clever, the result is generally for the worse.

To assure that the above results are not specific to trace `emacs1`, we have duplicated the runs in Figure 10 on a quite different trace, `kestrel.mar1`; nearly ten hours long, this trace is on a workload including "software development, documentation, e-mail, simulation, and other typical activities of engineering workstations" [9]. The results are graphed in Figure 11. Since the trace-data identified much of the idle time in `kestrel.mar1` as soft, we were able to do these runs using the simulator option of stretching run_cycles only into soft_idle. Also note that delay factors are unusually small for this trace, apparently because a long block of full-run intervals, which our policies handle near-optimally, dominates the delay measure.

In spite of these substantial differences, comparative results for the various policies are similar to those on `emacs1`. The main difference is that PEAK 0.2 has edged ahead of FLAT 0.4 and LONG_SHORT 3 to become the optimal algorithm in the medium-delay range. It is notable that PEAK, having been designed to work well with thin peaks, proves particularly effective for small interval_lengths, at which such bursty peaks are common. Figure 12 illustrates the superior behavior of PEAK 0.2 relative to PAST by tracking the speeds they respectively set on a stretch of `kestrel.mar1`. These speeds are graphed along with the effective run_percent—that is, run_cycles divided by (run_cycles + soft_idle). Note the comparative smoothness and the greater correspondence to run_percent of the PEAK speeds.

6 Conclusions and directions for future research

We found that several of our predictive algorithms performed poorly; only PEAK exhibited strong performance. We might then conclude that simple algorithms based on rational smoothing rather than "smart" predicting may be most effective.

Nevertheless, further possibilities for prediction remain to be tried. A policy might sort past information by process-type; it could then use its knowledge of the expected run-times of various types of processes to better predict the system's near-future computational needs. Moreover, each application could provide the system with useful information—stating how much it expects to load the system in the near future and how long a delay of a given process it would regard as acceptable. Indeed, communication of straightforward deadlines to the system (a keystroke must be processed in 0.01 seconds, and so on) would be an obvious component of an optimal speed-setting policy.

Testing out such theories, however, would quickly go beyond the limits of a simulation. How soon, then, may we employ actual variable-speed systems?

CPUs already exist which are certified to operate over a range of possible voltages. The ARM Processor may operate at 2.5–3.6 V; while "a Motorola CMOS 6805 microcontroller (cloned by SGS-Thomson) is rated at 6 Mhz at 5.0 Volts, 4.5 Mhz at 3.3 Volts, and 3 Mhz at 2.2 Volts. This is a close to linear relationship between voltage and clock rate" [9]. Thus there is seemingly no technical objection to designing a variable-voltage system, provided that the input reference voltage to the proces-

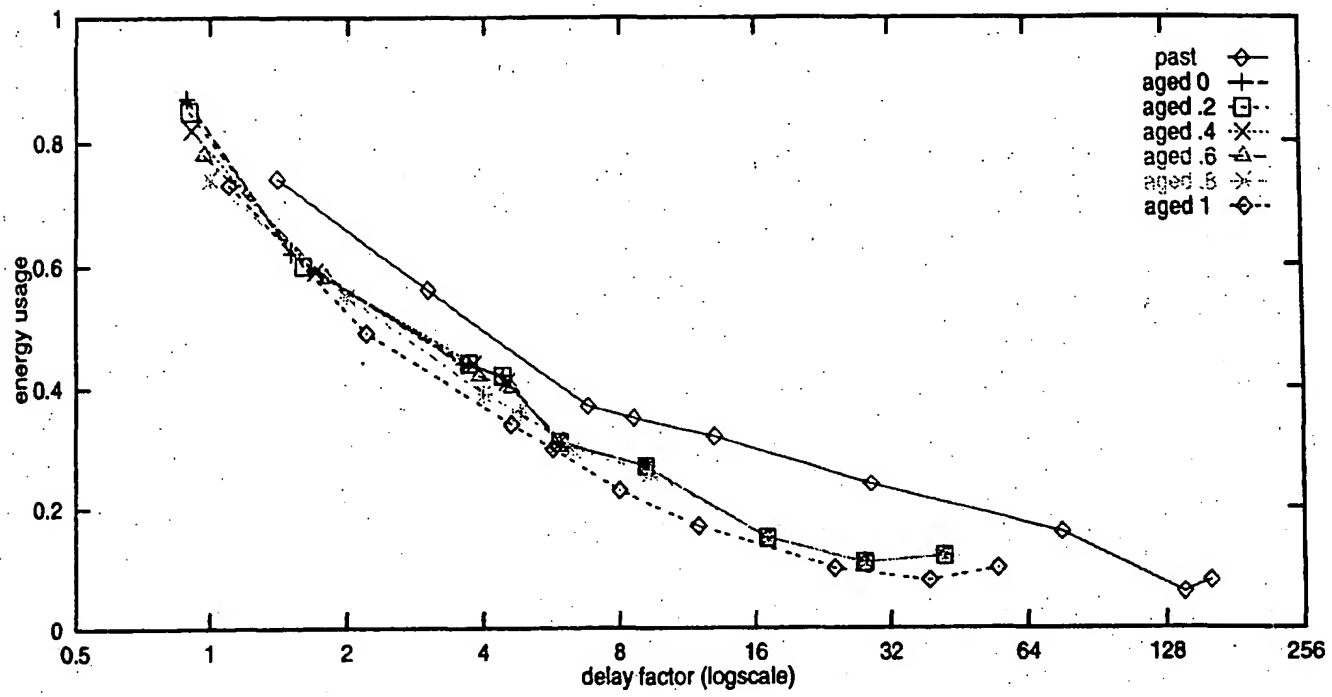

Figure 6: Performance of policy AGED_AVERAGES on trace `emacs1`. AGED_AVERAGES, run with several possible values of its parameter ⟨*const*⟩, is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

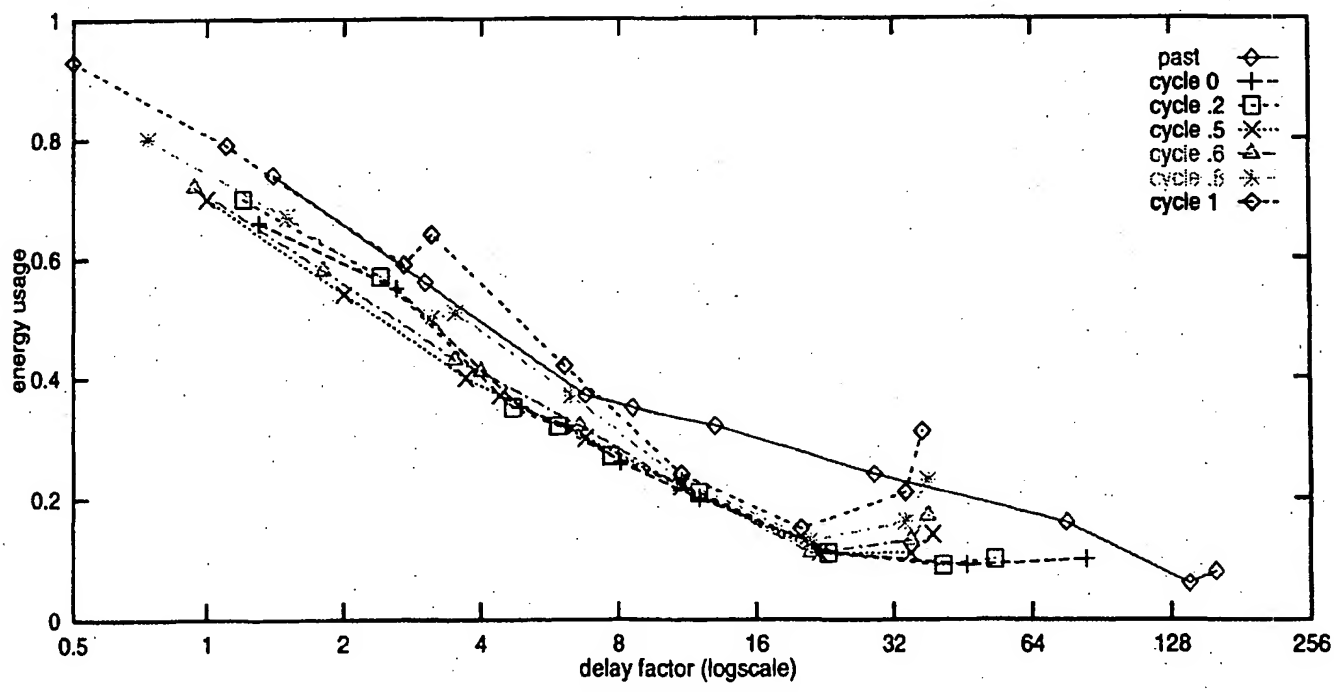

Figure 7: Performance of policy CYCLE on trace `emacs1`. CYCLE, run with several possible values of its parameter (*const*), is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

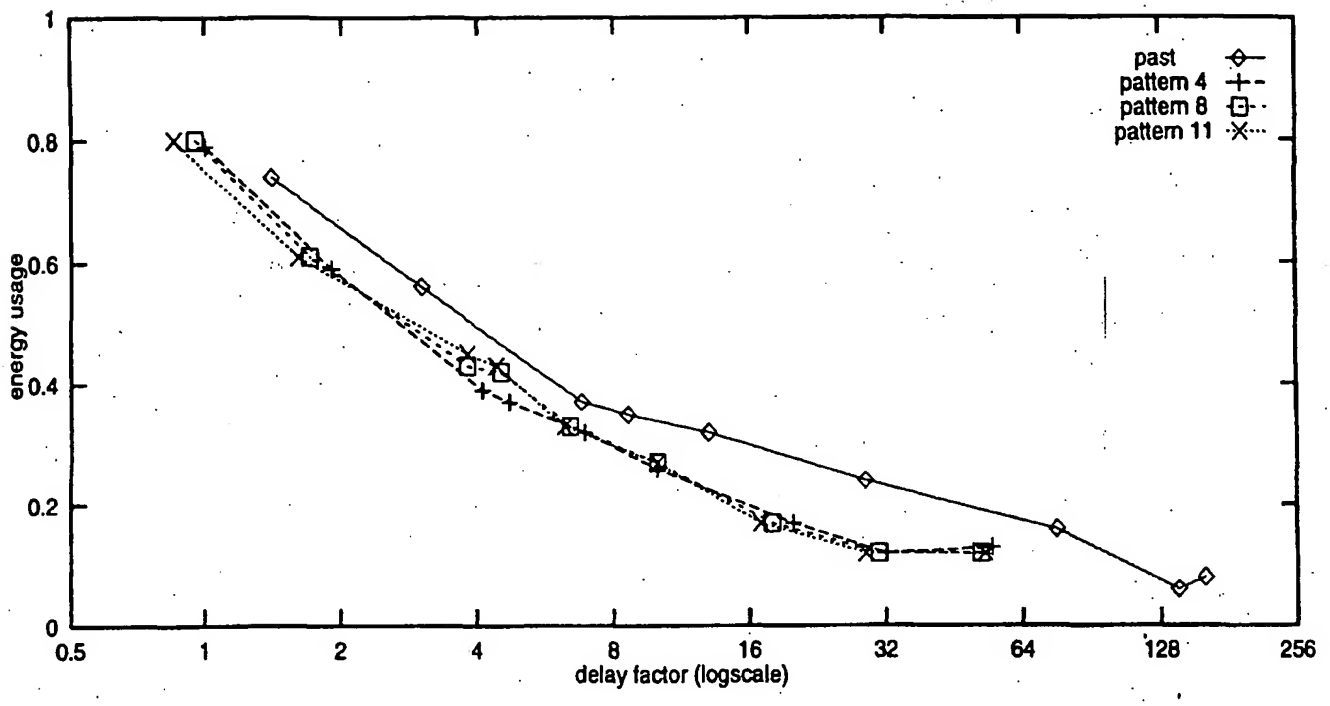

Figure 8: Performance of policy PATTERN on trace emacs1. PATTERN, run with several possible values of its parameter (*const*), is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

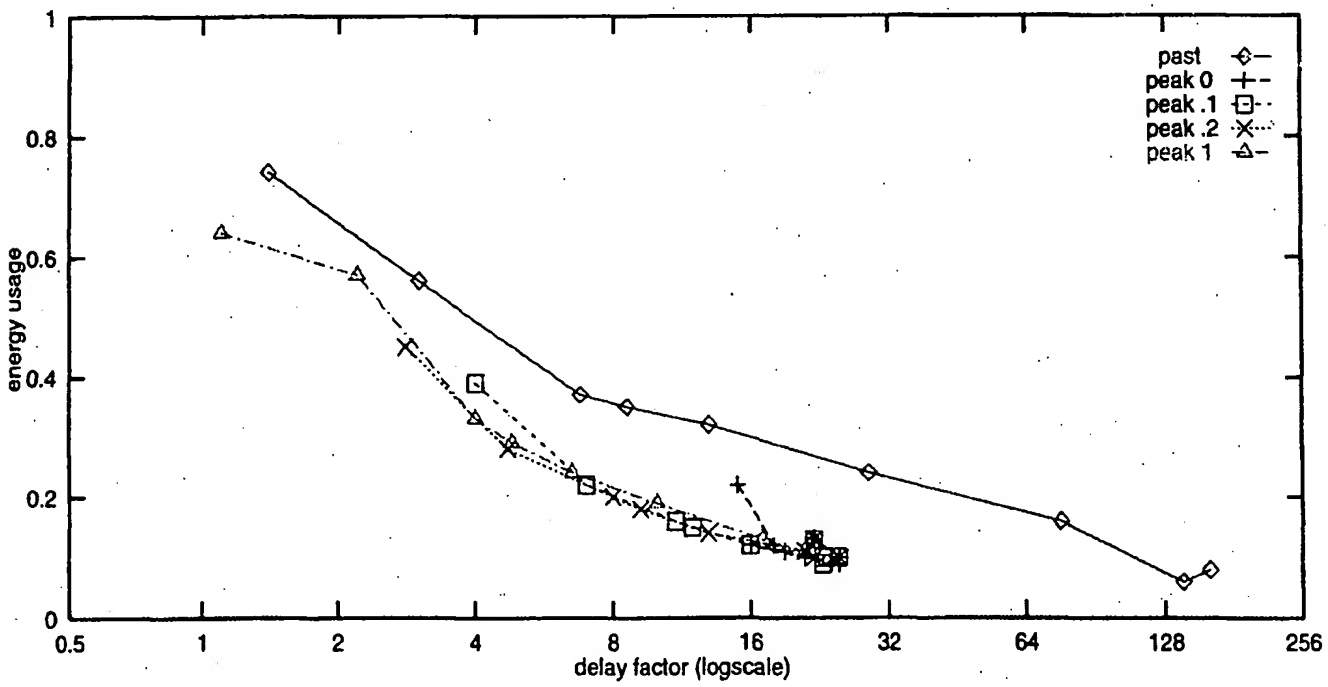

Figure 9: Performance of policy PEAK on trace emacs1. PEAK, run with several possible values of its parameter (*const*), is compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

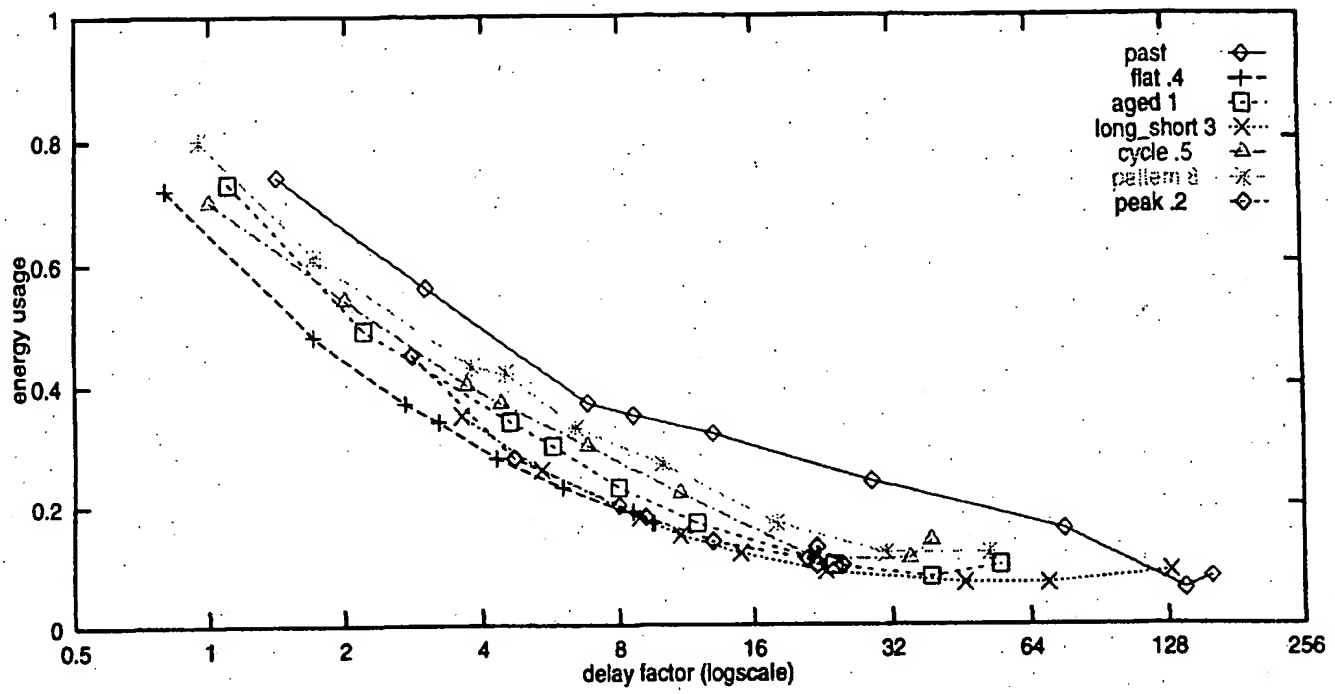

Figure 10: Performance of various policies on trace emacs1. The best policies from Figures 4–9 are here compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

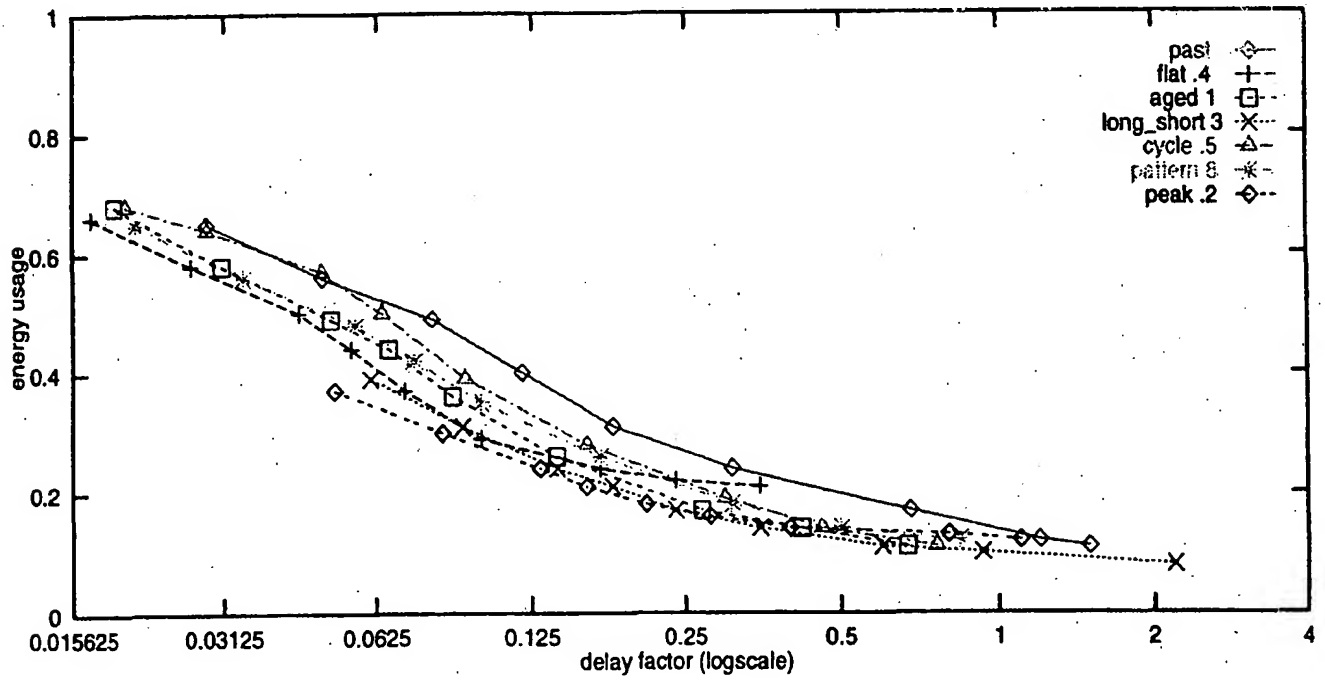

Figure 11: Performance of various policies on trace kestrel.mar1. The best policies from Figures 4–9 are here compared to PAST. For each policy, interval_lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

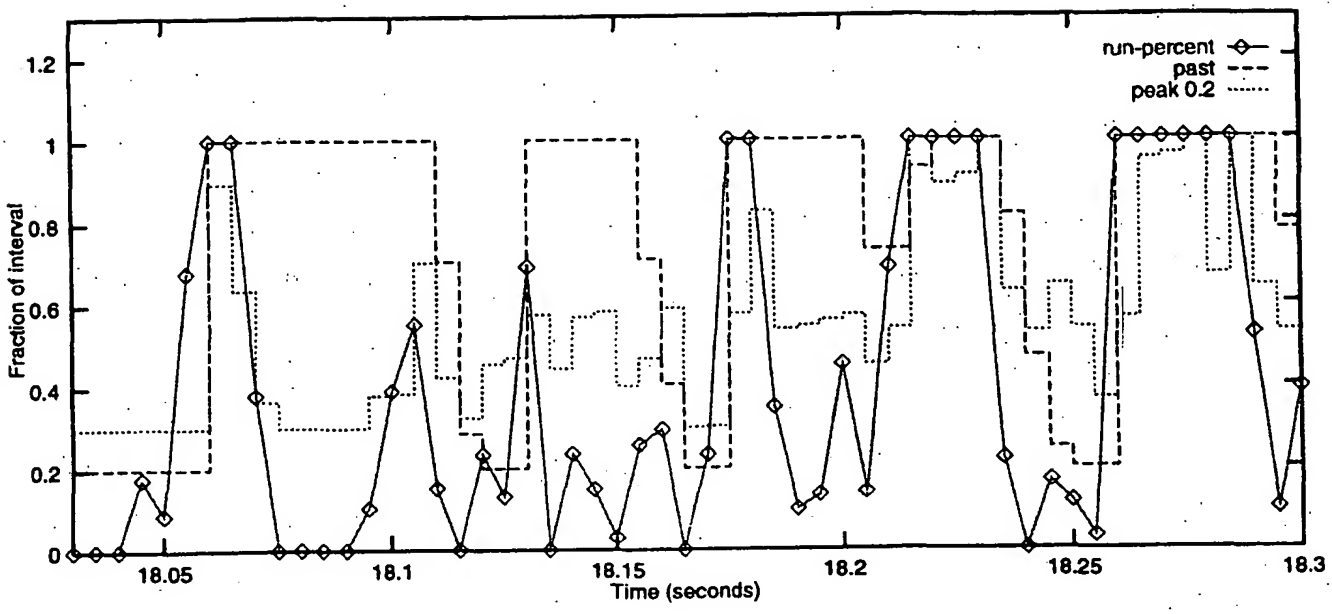

Figure 12: A stretch of the `kestrel.mar1` trace, with effective run_percent (= run_cycles / (run_cycles + soft_idle)) graphed alongside the resulting speeds set by PEAK 0.2 and by PAST. Interval_length is 0.005 seconds.

sor's voltage regulator may be a digital word writable by the processor.

The main time-cost would be for the converter to ramp the supply voltage and for the processor's phase-locked loop, if present, to change clock frequency. Thus ramping-time is determined by the time-constants of the converter and the phase-locked loop, and so would be on the order of tens of μsec [9, 5]. This time-scale seems well suited to our policies, which allow $\geq$ 5000 μsec between speed changes. Moreover, the CPU should be able to continue working during a voltage ramp; and ramping should not have any substantial power-cost.

Our results defer, then, to hopefully imminent studies of actual systems.

We thank Marvin Theimer, Mark Weiser, Alan Demers, Scott Shenker, Thomas Burd, and particularly Brent Welch, without whose help this project would not have been possible.

References

[1] T.D. Burd and R.W. Brodersen, "Energy efficient CMOS microprocessor design," *Proc. 28th Hawaii Int'l Conf. on System Sciences*, Vol. 1, pp. 288–297, Jan. 1995.

[2] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, Vol. 27, pp. 473–484, Ap. 1992.

[3] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the power-hungry disk," *Proc. Winter 1994 USENIX Conf.*, pp. 293–306, Jan. 1994.

[4] M.A. Horowitz, "Self-clocked structures for low power systems," ARPA semi-annual report, Computer Systems Laboratory, Stanford University, Dec. 1993.

[5] M. Ishigami, "Pentium Processor (610\75) design considerations for mobile systems," *Intel Application Note AP-518*, http://techdoc.wais.net:2160/default.html, Oct. 1994.

[6] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A quantitative analysis of disk drive power management in portable computers," *Proc. Winter 1994 USENIX Conf.*, pp. 279–292, Jan. 1994.

[7] K. Li, "Towards a low power file system," *CS Tech. Report 94-814*, University of California, Berkeley, May 1994.

[8] M. Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM*, Vol. 36, pp. 74–83, July 1993.

[9] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," *Proc. 1st USENIX Symp. on Operating Systems Design and Implementation*, pp. 13–23, Nov. 1994.